A Pattern-Based Comparison of OpenACC & OpenMP for Accelerators

Sandra Wienke\textsuperscript{1,2}, Christian Terboven\textsuperscript{1,2}, James C. Beyer\textsuperscript{3}, Matthias S. Müller\textsuperscript{1,2}

\textsuperscript{1}IT Center, RWTH Aachen University
\textsuperscript{2}JARA-HPC, Aachen
\textsuperscript{3}Cray Inc.
Agenda

- Motivation
- Overview on OpenACC & OpenMP Device Capabilities
- Expressiveness by Pattern-Based Comparison
  - Map
  - Stencil
  - Fork-Join
  - Superscalar Sequence
  - Parallel Update
- Summary
- Conclusion & Outlook
Motivation

- Today/ future: Heterogeneity & specialized accelerators
  - Consequence: Parallel programming gets more complex
- Directive-based programming models may simplify software design
- Two well-promoted approaches: OpenACC, OpenMP

- User question: Which programming model to use?
  - Power/ expressiveness, performance, long-term perspective
  - Targeted hardware: GPU, Xeon Phi,…

- Expressiveness by pattern-based approach
  - Patterns: basic structural entities of algorithms
  - Used: structured parallel patterns [1], e.g. map, stencil, reduction, fork-join, superscalar sequence, nesting, parallel update, and geometric decomposition

# Overview on OpenACC & OpenMP Device Capabilities

## Directive-based accelerator programming: C/C++, Fortran

### OpenACC
- Initiated by Cray, CAPS, PGI, NVIDIA
- 2011: specification v1.0
- 2013: specification v2.0

### OpenMP
- Broad (shared-memory) standard
- 2013: specification v4.0

## Comparison of Constructs

<table>
<thead>
<tr>
<th>OpenACC</th>
<th>OpenMP</th>
<th>Description of constructs/ clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>parallel</td>
<td>target</td>
<td>offload work</td>
</tr>
<tr>
<td>parallel</td>
<td>teams, parallel</td>
<td>create in par. running threads</td>
</tr>
<tr>
<td>kernels</td>
<td></td>
<td>compiler finds parallelism</td>
</tr>
<tr>
<td>loop</td>
<td>distribute, do, for, simd</td>
<td>worksharing across parallel units</td>
</tr>
<tr>
<td>data</td>
<td>target data</td>
<td>manage data transfers (block)</td>
</tr>
<tr>
<td>enter data</td>
<td></td>
<td>unstructured data lifetime: to and from the device</td>
</tr>
<tr>
<td>exit data</td>
<td></td>
<td>data movement in data environment</td>
</tr>
<tr>
<td>update</td>
<td>target update</td>
<td>interoperability with CUDA/ libs</td>
</tr>
<tr>
<td>host data</td>
<td></td>
<td>advice to put objects to closer memory</td>
</tr>
<tr>
<td>cache</td>
<td></td>
<td>strip-mining of data</td>
</tr>
<tr>
<td>tile</td>
<td></td>
<td></td>
</tr>
<tr>
<td>declare</td>
<td>declare target</td>
<td>declare global, static data</td>
</tr>
<tr>
<td>routine</td>
<td>declare target</td>
<td>for function calls</td>
</tr>
<tr>
<td>async(int)</td>
<td>task depend</td>
<td>async. exec. w/ dependencies</td>
</tr>
<tr>
<td>wait</td>
<td>taskwait</td>
<td>sync of streams/ tasks</td>
</tr>
<tr>
<td>async wait</td>
<td></td>
<td>async. waiting</td>
</tr>
<tr>
<td>parallel in parallel</td>
<td>parallel in parallel/ team</td>
<td>nested parallelism</td>
</tr>
<tr>
<td>device_type</td>
<td></td>
<td>device-specific clause tuning</td>
</tr>
<tr>
<td>atomic</td>
<td></td>
<td>atomic operations</td>
</tr>
<tr>
<td>sections</td>
<td></td>
<td>non-iterative worksharing</td>
</tr>
<tr>
<td>critical</td>
<td></td>
<td>block executed by master</td>
</tr>
<tr>
<td>master, single</td>
<td></td>
<td>block executed by one thread</td>
</tr>
<tr>
<td>barrier</td>
<td></td>
<td>synchronization of threads</td>
</tr>
</tbody>
</table>

We talk about the following specifications: OpenACC 2.0, OpenMP 4.0.
Overview on OpenACC & OpenMP
Device Capabilities

**OpenACC**

**Execution & Memory Model**

Host-directed execution
Weak device memory model (no sync at gang/team level)
Separate or shared memory

**OpenMP**

We talk about the following specifications: OpenACC 2.0, OpenMP 4.0.
Map

- Maps in parallel different elements of input data to output collection
- Elemental function: $B = p \cdot A^T$

```c
#pragma acc routine seq
double f(double p, double aij) {
    return (p * aij);
}
// [...]
#pragma acc parallel
#pragma acc loop gang vector
for(i=0; i<n; i++) {
    // [...]
    #pragma acc loop vector
    for(j=0; j<m; j++) {
        b[j][i] = f(5.0, a[i][j]);
    }
}
```

OpenACC

```c
#pragma omp declare target
double f(double p, double aij) {
    return (p * aij);
}
#pragma omp end declare target
// [...]
#pragma omp target
#pragma omp teams distribute
for(i=0; i<n; i++) {
    // [...]
    #pragma omp parallel for
    for(j=0; j<m; j++) {
        b[j][i] = f(5.0, a[i][j]);
    }
}
```

OpenMP

- **Parallelism + workshare**
  - Kernels, parallel, loop
- **Levels of parallelism**
  - Automatic; tuning: gang/worker/vector
  - Performance portability possible
- **routine for function calls**
  - Denote parallelism: tuning, but call from same context

**OpenACC & OpenMP for Accelerators**

Sandra Wienke | IT Center der RWTH Aachen University

Stencil

- Access of neighboring input elements with fixed offsets
- Ability for data reuse & cache optimization

```c
#pragma acc parallel
#pragma acc loop tile(64,4) gang vector
for(i=1; i<n-1; i++) {
    for(j=1; j<m-1; j++) {
        #pragma acc cache(a[i-1:3][j-1:3])
        anew[i][j] = (a[i-1][j] + a[i+1][j] + a[i][j-1] + a[i][j+1]) * 0.25;
    }
}
```

- **tile**: strip-mining
  - 1st no. = block size of most inner loop
- **cache**: data caching
  - Just hint, compiler can ignore
  - (Performance-wise) needed especially for software-managed mem (GPUs)

```c
#pragma omp target
#pragma omp teams distribute collapse(2)
for(i=1; i<n-1; i+=4) {
    for(j=1; j<m-1; j+=64) {
        #pragma omp parallel for collapse(2)
        for(k=i; k<min(n-1,i+4); k++) {
            for(l=j; l<min(m-1,j+64); l++) {
                anew[k][l] = (a[k-1][l] + a[k+1][l] + a[k][l-1] + a[k][l+1]) * 0.25;
            }
        }
    }
}
```

- Tiling must be expressed explicitly
  - More development effort than w/ tiling
  - Performance expected similar to tiling
- No caching hints possible
  - Maybe performance loss on GPUs

Fork-Join

- Workflow is split (forked) into parallel/independent flows
- Merged (joined) again later

---

OpenACC

// no tasking concept

- Data parallelism possible
- Task parallelism
  → Just approximation by host-directed nested parallel constructs

---

OpenMP

```c
#pragma omp declare target
int fib(int n) {
  int x, y;
  if (n < 2) {return n;}
  #pragma omp task shared(x)
  x = fib(n - 1);
  #pragma omp task shared(y)
  y = fib(n - 2);
  #pragma omp taskwait
  return (x+y);
}
#pragma omp end declare target
```

---

- Data parallelism possible
- Task parallelism possible
  → sections, tasks

---

Superscalar Sequence

- Parallelization of ordered list of tasks
- Satisfaction of data dependencies
- Additional interpretation: heterogeneity

```
#pragma acc parallel loop async(1)
// F = f(A)
#pragma acc parallel loop async(2)
// G = g(B)
#pragma acc wait(1,2) async(3)
#pragma acc parallel loop async(3)
// H = h(F,G)
#pragma acc wait(1)
// S = s(F)
#pragma acc wait
```

- Streaming concept
  → async(int) + wait async

- Calling thread
  asynchronously continues

```
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(out:F)
  #pragma omp task depend(out:G)
  #pragma omp task depend(in:F,G) depend(inout:H)
  #pragma omp taskwait
}
```

- Tasking concept
  → Task dependencies
Parallel Update (1/3)

- **Own extension of pattern definitions**
- **Synchronization of data between (separate) memories**

---

**Both: data movement**
- Map data to/from device, allocate
- Data regions, update constructs

```c
void stencilOnAcc(double **a, double ** anew, int n, int m) {
    #pragma acc parallel present 
    (a[1:n-2][0:m], anew[1:n-2][0:m])
    #pragma acc loop // stencil comp.
} // [..]
#pragma acc data create(Anew[0:n][0:m]) \ 
    copyin(A[0:n][0:m])
{
    while (iter < iter_max) {
        stencilOnAcc(A,Anew,n,m);
        #pragma acc update host(Anew[1:n-2][0:m])
        swapOnHost(A,Anew,n,m);
        #pragma acc update device(A[1:n-2][0:m])
        iter++;
    }
}
```

---

```c
void stencilOnAcc(double **a, double** anew, int n, int m) {
    #pragma omp target map(tofrom: 
    (a[1:n-2][0:m], anew[1:n-2][0:m])
    #pragma omp teams distribute 
    parallel for // stencil comp.
} // [..]
#pragma omp target data 
    map(alloc:Anew[0:n][0:m]) \ 
    map(to:A[0:n][0:m])
{
    while (iter < iter_max) {
        stencilOnAcc(A,Anew,n,m);
        #pragma omp target update \ 
        from(Anew[1:n-2][0:m])
        swapOnHost(A,Anew,n,m);
        #pragma omp target update \ 
        to(A[1:n-2][0:m])
        iter++;
    }
}
```
Parallel Update (2/3)

- Own extension of pattern definitions
- Synchronization of data between (separate) memories

```c
void stencilOnAcc(double **a, double ** anew, int n, int m) {
    #pragma acc parallel present (a[1:n-2][0:m], anew[1:n-2][0:m])
    #pragma acc loop // stencil comp.
} // [..]
#pragma acc data create(Anew[0:n][0:m]) \
    copyin(A[0:n][0:m]) if(test)
{
    while (iter < iter_max) {
        stencilOnAcc(A,Anew,n,m);
        #pragma acc update host(Anew[1:n-2][0:m])
        swapOnHost(A,Anew,n,m);
        #pragma acc update device(A[1:n-2][0:m])
        iter++;
    }
}
```

- Missing data: Error
  - Present: no data movement; error if data not there

```c
void stencilOnAcc(double **a, double** anew, int n, int m) {
    #pragma omp target map(tofrom: 
        a[1:n-2][0:m], anew[1:n-2][0:m])
    #pragma omp target data \ 
        map(alloc:Anew[0:n][0:m]) \ 
        map(to:A[0:n][0:m]) if(test)
{
    while (iter < iter_max) {
        stencilOnAcc(A,Anew,n,m);
        #pragma omp target update \ 
            from(Anew[1:n-2][0:m])
        swapOnHost(A,Anew,n,m);
        #pragma omp target update \ 
            to(A[1:n-2][0:m])
        iter++;
    }
}
```

- Missing data: Fix-Up
  - Implicit check for data; move if not there
Parallel Update (3/3)

- Own extension of pattern definitions
- Synchronization of data between (separate) memories

```cpp
class CArray {
public:
    CArray(int n) {
        a = new double[n];
        #pragma acc enter data create(a[0:n])
    }
    ~CArray() {
        #pragma acc exit data delete(a[0:n])
        delete(a);
    }
    void fillArray(int n) {
        #pragma acc parallel loop
        for(int i=0; i<n; i++) { a[i]=i; }
    }
private:
    double *a;
};
```

- Unstructured data lifetimes
  → enter/exit data
  → API functions also exist

- Use cases
  → init/fini functions or constructors / destructors
  → Manual deep copies of pointer structures
  → OpenMP committee is working on it
# Summary

## Parallel Pattern

<table>
<thead>
<tr>
<th>OpenACC</th>
<th>OpenMP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Map</strong></td>
<td><strong>F (kernels, parallel, loop, gang/worker/vector, routine)</strong></td>
</tr>
<tr>
<td><strong>Stencil</strong></td>
<td><strong>I (tile, cache)</strong></td>
</tr>
<tr>
<td><strong>Reduction</strong></td>
<td><strong>F (reduction)</strong></td>
</tr>
<tr>
<td><strong>Fork-Join</strong></td>
<td><strong>F (tasks)</strong></td>
</tr>
<tr>
<td><strong>Workpile</strong></td>
<td><strong>F (tasks)</strong></td>
</tr>
<tr>
<td><strong>Superscalar Sequence</strong></td>
<td><strong>F (tasks)</strong></td>
</tr>
<tr>
<td><strong>Nesting</strong></td>
<td><strong>F (parallel in parallel)</strong></td>
</tr>
<tr>
<td><strong>Parallel Update</strong></td>
<td><strong>F (copy, data region, update, enter/exit data, error concept)</strong></td>
</tr>
<tr>
<td><strong>Geometric Decomposition</strong></td>
<td><strong>F (acc_set_device, device_type)</strong></td>
</tr>
</tbody>
</table>

**F**: supported directly, with a special feature/ language construct  
**F-**: base aspects (but not all) directly supported  
**I**: implementable easily and efficiently using other features  
**-**: not implementable easily


---

**OpenACC & OpenMP for Accelerators**  
**Sandra Wienke** | IT Center der RWTH Aachen University
Conclusion & Outlook (1/2)

- **Concurrently: OpenACC one step ahead of OpenMP**
  - Less general concepts, but more features
  - Today’s recommendation for GPUs: OpenACC
    - Port to OpenMP will be easy
      - (with OpenACC 1.0 features, or new OpenMP features)

- **Long-term perspective: Development of standards**
  - Our assumption (OpenACC, OpenMP committee work): co-existence
  - OpenMP might be advantageous in the long term
    - Broader user / vendor base (portability)
  - But, implementation effort significant
    - Might imply limited OpenMP functionality within offload regions
Conclusion & Outlook (2/2)

Long-term perspective: architectures
- Possible shared memory by host & device: supported by both models
- But, promising perspectives for offload mode in general?
  - OpenMP might take lead with non-offload features

Long-term perspective: performance
- We don’t expect differences between equivalent approaches (same target hardware, compiler)
- Dissimilarities might occur if general concept differs (e.g. streams vs. tasks)
- Performance evaluation is future work (including comparison to other models)

Thank you for your attention!